TUBITAK-UZAY Compulsory Summer Internship Project

FINAL DELIVERY REPORT

TUBITAK-UZAY-MultiCommSim

2025 SUMMER

EDİZ ARKIN KOBAK – Computer Engineer

TABLE OF CONTENTS

1. Introduction

Bu proje, TÜBİTAK UZAY'ın uydu simülasyon cihazlarında kullanılmak üzere geliştirilen bir haberleşme altyapısını temsil etmektedir. Amaç, yalnızca **tek bir IP ve port** üzerinden **birden fazla client-server çiftinin**, **Docker ile izole edilmiş şekilde**, eş zamanlı TCP mesajlaşmasını sağlamaktır. Sistem; Java ile geliştirilmiş Router, Client ve Server uygulamalarından, React tabanlı frontend'den ve Flask-Python API'lerinden oluşmaktadır.

2. Requirements and Solutions

FR Kodu	Gereksinim Açıklaması	Nasıl Karşılandı
FR-1	Tek IP & Tek Port Üzerinden Çoklu Bağlantı	Tüm client'lar sadece **router:6003** üzerinden bağlanıyor. RouterService gelen bağlantılardan clientID çıkararak oturumları izole ediyor. Aynı porttan gelen tüm bağlantılar ayrı thread'ler üzerinden eşzamanlı işleniyor.
FR-2	Docker İçinde Server Zorunluluğu	Her Server uygulaması, docker run komutu ile ayrı container'da başlatılıyor (java-message-server).
FR-3	Client Docker Bağımsızlığı	Client uygulamaları hem container içinde hem dış CLI'dan çalıştırılabilir. UI, Flask üzerinden peer başlatabiliyor.
FR-4	Docker Üzerinden Yönetim	Flask üzerinden yapılan her çağrı, docker-py ile container'ları programatik olarak oluşturuyor. Komut satırı kullanımı gerekmiyor.
FR-5	TCP ile Veri Alışverişi	Tüm istemci ve sunucular TCP üzerinden iletişim kuruyor. Mesajlar Gson kullanılarak **JSON** formatına çevriliyor.

FR Kodu	Gereksinim Açıklaması	Nasıl Karşılandı
FR-6	Bağlantı İzolasyonu	Her peer (client-server çifti) yalnızca kendi container'ına yönlendiriliyor. Diğer oturumlarla veri karışıklığı olmuyor.
FR-7	Oturum Yönlendirme	SessionManager sınıfı, gelen clientId'ye karşılık gelen server container'ı tanımlıyor ve yönlendirme yapıyor.
FR-8	Çoklu Bağlantı Performansı	10'dan fazla eş zamanlı peer oluşturulup başarıyla test edildi. Router her biri için ayrı thread ile çalıştı.
FR-9	Port Çoklama Alternatif	Dinamik port ataması yapılmadı. Tek port üzerinden routing çözümü uygulandı.
FR-10	Loglama	Her bir container'ın logu *container.logs()* ile Flask üzerinden okunuyor ve React UI'da gösteriliyor.

---

3. Backend Architecture

3.1 Architectural Layers

Katman	Teknoloji	Görev
Application Layer	Java	Client & Server TCP mesajlaşması
Router Layer	Java	Tüm mesajların alınıp oturumlara ayrılması
Container Layer	Docker	Her peer izole olarak burada çalışır
API Layer	Python (Flask)	UI ile backend arasında köprü
UI Layer	React	Arayüz: peer oluşturma, log gösterimi
Testing Layer	Python	Otomatik test komutları
Deployment Layer	Docker CLI	Peer başlatma ve sonlandırma işlemleri

3.2 Detailed Backend Architecture and Components

Bu bölümde MultiCommSim sisteminin tüm backend bileşenleri detaylı olarak açıklanmıştır. Uygulama üç temel backend bileşeninden oluşur:

1. **Router Servisi (Java)**
2. **Server Container (Java)**
3. **Client Container (Java)**

Router servisi, tüm TCP bağlantılarını yönetir; client'ların gönderdiği mesajları hedef sunuculara yönlendirir. Flask API ise bu sistemin dış arayüzüdür; Docker container'larının yaratılması, testlerin çalıştırılması gibi kontrol işlemlerini sağlar. React ise frontend'dir ve logların görüntülenmesini sağlar.

3.2.1 Communication Flow

i. Kullanıcı, React arayüzü üzerinden yeni bir peer yaratır (create-peer çağrısı).
ii. Flask API, Docker ile bir client ve bir server container'ı başlatır.
iii. Server container, **6003** portu üzerinde dinleme yapar.
iv. Client container, router'a TCP üzerinden bağlanır, mesajını yollar.
v. Router, mesajı ilgili server container'a iletir.
vi. Server cevabı router'a yollar → router cevabı client'a iletir.
vii. Client container görevi bitince kapanır. Server container açık kalır.
viii. Bu süreçte aşağıdaki bileşenler görev alır.

3.2.2 Class and Function Descriptions

Router Side

❖ **RouterService.java**

Router'ın giriş noktasıdır.

- Ana *main()* fonksiyonu burada çalışır.
- TCPListener sınıfını başlatır.
- İleride WebSocket desteği eklenirse HTTP WebSocket endpoint'i de buradan yönetilecektir.
- Gerekirse router UI'ye (React) canlı log göndermek için WebSocket açabilir.

❖ TCPListener.java

- ServerSocket üzerinden gelen client bağlantılarını dinler.
- Her bağlantı için ayrı bir Thread başlatır.
- Thread içinde gelen mesaj MessageRouter üzerinden işlenir.

❖ MessageRouter.java

- Router'ın mesaj yönlendirme motorudur.
- Client'tan gelen **JSON** mesajı parse eder (Gson).
- Mesajda hedef sunucu adı (targetIp) ve içerik (message) vardır.
- TCPConnectionPool kullanarak ilgili server'a TCP bağlantısı yapar.
- Mesajı gönderir, cevabı alır, client'a döner.

❖ TCPConnectionPool.java

- Daha önce bağlantı yapılmış bir server'a yeniden TCP bağlantısı açılmasını engeller.
- Aynı IP'ye birden fazla bağlantı yapılmaması için Map<String, Socket> yapısı tutar.
- Otomatik olarak bağlantıyı cache'ler, reuse eder.

❖ SessionManager.java

- Router tarafında client ve server bağlantılarını clientId ↔ Socket şeklinde eşleyerek aktif olarak session yönetimini sağlar.
- Her client'ın bağlantısı eşsiz bir clientId ile takip edilir ve bu sayede yönlendirici (Router) gelen mesajları doğru hedefe ulaştırabilir

Server Side

❖ ServerApp.java

- Container içinde çalışan TCP sunucu uygulamasıdır.
- **SERVER_PORT=6003** üzerinden dinleme yapar.
- *accept()* ile router'dan gelen bağlantıyı alır.
- Gelen mesajı BufferedReader ile okur.
- İlgili mesajı işler (şu an sadece echo) ve cevabı PrintWriter ile döner.

Client Side

❖ ClientApp.java

- Container içinde çalışır.
- Çevresel değişkenlerden şunları alır:
  - **ROUTER_HOST (varsayılan: router)**
  - **ROUTER_PORT (varsayılan: 6003)**
  - **CLIENT_ID**
  - **SERVER_HOST**
  - **CLIENT_MSG**
- Bu bilgilerle router'a bağlanır.
- Mesajı **JSON** formatında gönderir.
- Cevabı alır ve stdout'a yazdırır.
- Bağlantı sona erer → container otomatik kapanır.

Flask API (Python Backend)

create-peer (POST)

- clientMsg, serverMsg içeren bir JSON alır.
- uuid ile benzersiz peerId üretir.
- *docker_client.containers.run()* ile:
  - server-{peerId}
  - client-{peerId} container'ları yaratılır.
- Her container docker_simnet ağına bağlanır.
- Client container görevi bitince otomatik kapanır.

run-test (POST)

- Docker API ile tüm server-*, client-* container'ları listeler.
- Her birinin logunu .logs() ile okur.
- JSON dictionary formatında döndürür: {containerName: [log lines]}
- Sonrasında test için oluşturulan tüm container'ları .remove(force=True) ile siler.

4. Deviations & Parallels from Design

4.1 Removing the WebSocket plugin

Sebep:

WebSocket kullanımı için React UI'dan gelen mesajların router içinde TCP'ye dönüştürülmesi gerekiyordu. Ancak React UI doğrudan TCP bağlantı açamaz (browser policy). WebSocket desteği planlandı ama:

- **Öncelikli hedef TCP testleri ve container izolasyonuydu**
- **Flask API üzerinden test senaryoları çok daha hızlı yürütülebildi**

Alternatif Çözüm:

Flask API ile peer başlatma ve run-test çağrısı üzerinden TCP loglar toplanarak UI'da gösterildi. WebSocket'ten gelen veriler yerine Flask üzerinden polling yapıldı (/logs).

.

---

5. Port Management

MultiCommSim projesinde her server-client çifti (peer) aynı port üzerinden haberleşmektedir: **6003**. Bu port hem sunucu tarafında dinleme noktasıdır hem de istemci tarafında hedef bağlantı noktasıdır. Sistemin ölçeklenebilirliği ve dinamik container yönetimi göz önünde bulundurularak port yönlendirme (port mapping) mekanizmaları dikkatle tasarlanmıştır.

5.1 Docker Port Forwarding

Docker, container'ların dış dünya ile iletişim kurabilmesi için **port binding** (bağlama) işlemini kullanır:

HOST_PORT → CONTAINER_PORT

Bu yapı sayesinde, host makinede (örneğin Ubuntu sisteminde) **localhost:32866** adresinden gelen istekler container içinde çalışan bir uygulamanın dinlediği **6003** portuna yönlendirilir.

5.2 Single Port Usage and Routing Logic

MultiCommSim sisteminde, router servisinin dış dünya ile haberleştiği tek bir TCP portu vardır: **6003**. Bu port üzerinden tüm client'lar router'a bağlanır. Router ise bu veriyi container isimlerinden yola çıkarak ilgili sunucuya iletir.

Bu yaklaşım sayesinde:

- Tek bir router container'ı çalışır.
- Birden çok client container aynı porttan router'a ulaşabilir.
- Router, hedef sunucuyu mesaj içindeki **targetIp** veya **targetId** bilgisinden bulur.

5.3 EXPOSE and ports={...} Difference

Dockerfile içinde **EXPOSE 6003** tanımı yapılırsa bu sadece container'ın **6003** portunu açtığını belirtir. Ancak dış dünyaya açmak için **docker run** komutunda:

```
ports={"6003/tcp": None}
```

kullanılır. Burada None diyerek Docker'ın **rastgele bir host port** atamasına izin verilir.

Bu sayede aynı anda birden fazla server container başlatılabilir. Çünkü her biri için host port farklı olacaktır.

Örnek:

Container Adı	Container Port	Host Port (otomatik)
server-a1b2c3d4	**6003**	**32866**
server-e5f6g7h8	**6003**	**32868**

5.4 Multiple Peer Management from a Single Port: TCP Multiplexing

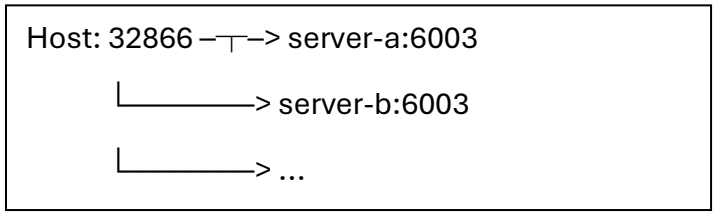
Sistem, tek porttan (**6003**) tüm client-server eşleşmelerini **RouterService** üzerinden yöneterek TCP multiplexing yapar.

1. Client **client-xxxx** olarak bağlanır, targetIp: **server-xxxx** şeklinde mesaj gönderir.
2. Router bu ID'ye göre bağlantı yönlendirir.
3. TCPConnectionPool sınıfı, hedef sunucuya socket yaratır veya mevcut olanı kullanır.
4. Böylece tüm peer'ler için farklı IP yok, tek bir router IP ve port (**6003**) vardır.

5.5 Docker Virtual Port Concept

Docker'da container'lar kendi izolasyon alanlarında çalıştığından, aynı container portu (**6003**) her bir container için tekrar tekrar kullanılabilir. Host → Container yönlendirmesi ise dışarıdan erişimi sağlar.

Bu yönlendirme şeması:



Her container **6003**'ü dinler ama dışarıya farklı porttan açılır.

5.6 Usage Scenario on a Real Device

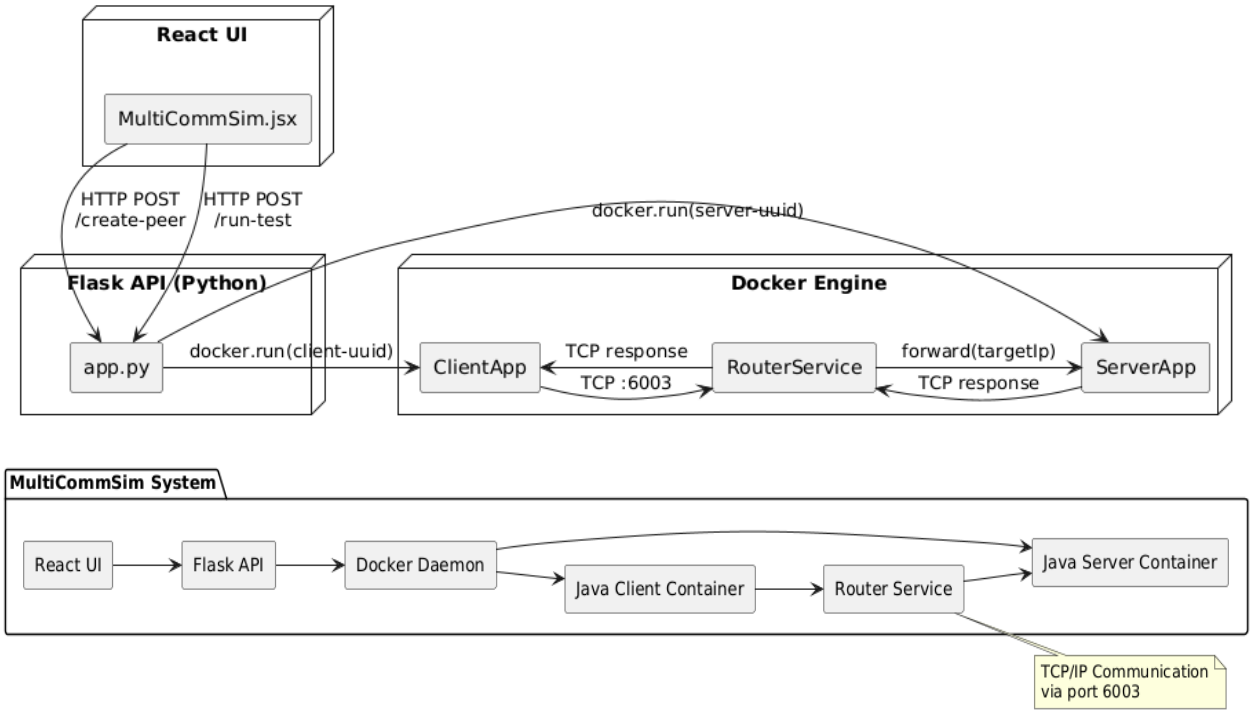
Eğer sistem bir tek fiziksel cihaza (örneğin bir uydu işlemcisine) yüklenirse:

- Tüm uygulamalar **aynı IP** üzerinde çalışır.
- **Router servisi tek bir port** (**6003**) üzerinden dış dünyadan mesaj alır.
- Server uygulamaları container içinde çalıştığından **6003**'ü dinlemeye devam edebilir.
- Dış dünya **client → router:6003** mesaj gönderir.
- Docker burada adeta **virtual IP + port routing mekanizması** gibi çalışır.

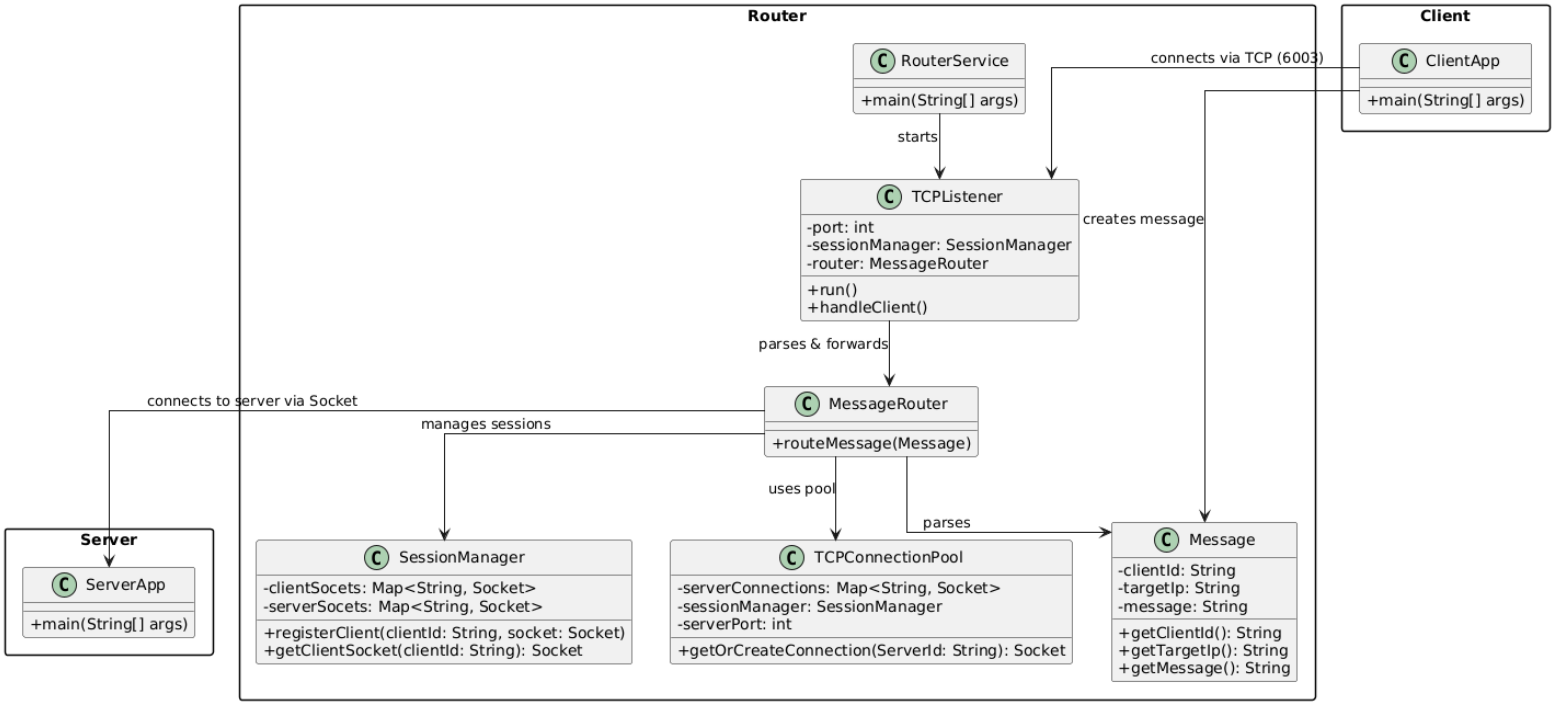
Bu nedenle bu mimari, tek port üzerinden sınırsız sayıda peer'in yönetilmesini sağlar. Bu durum **gerçek donanım üzerinde ölçeklenebilir dağıtık haberleşme** için oldukça değerlidir.

6. Diagrams and Images

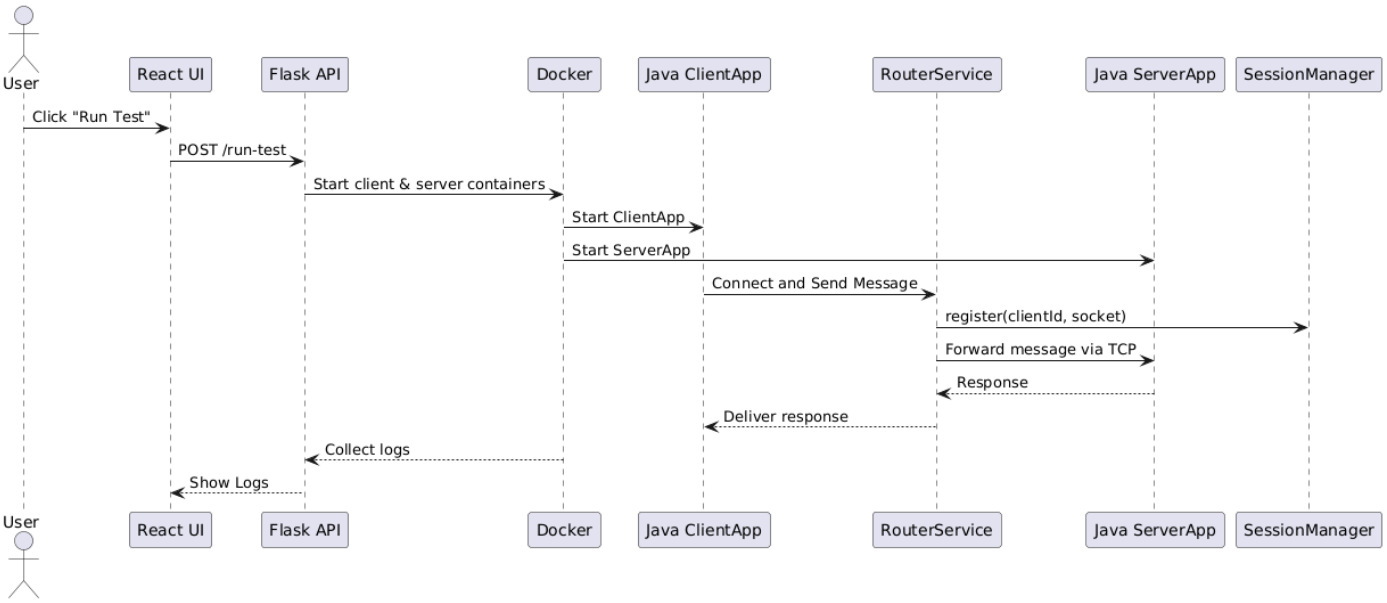
6.1 High Level Architecture



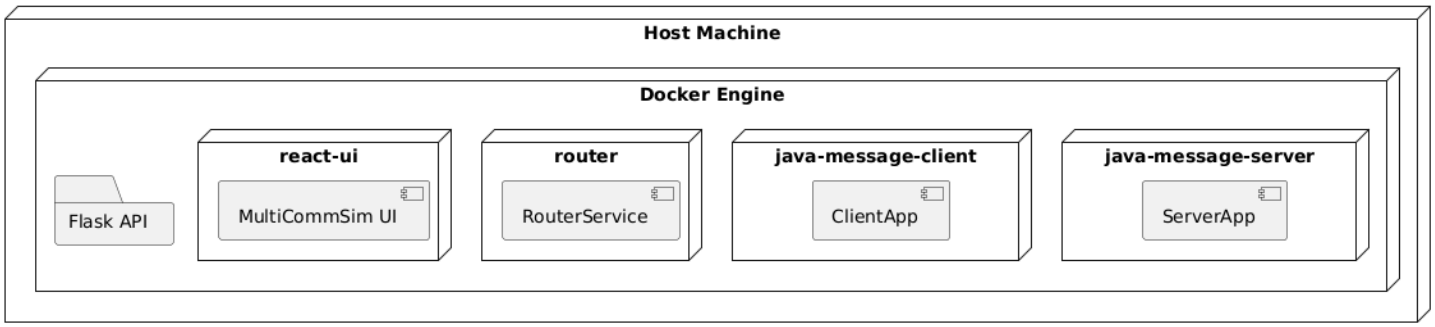
6.2 Backend Class Diagram



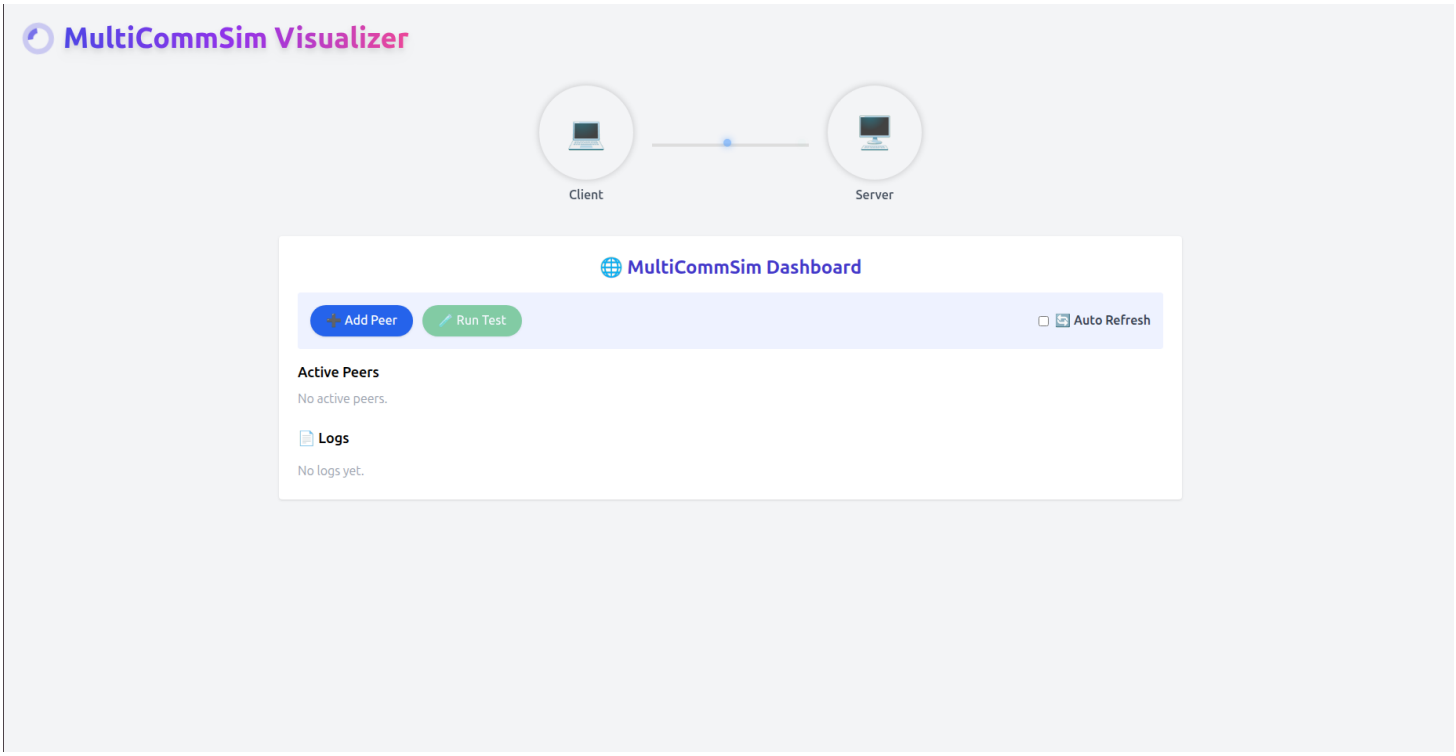
6.3 Client-Server Messaging Sequence Diagram

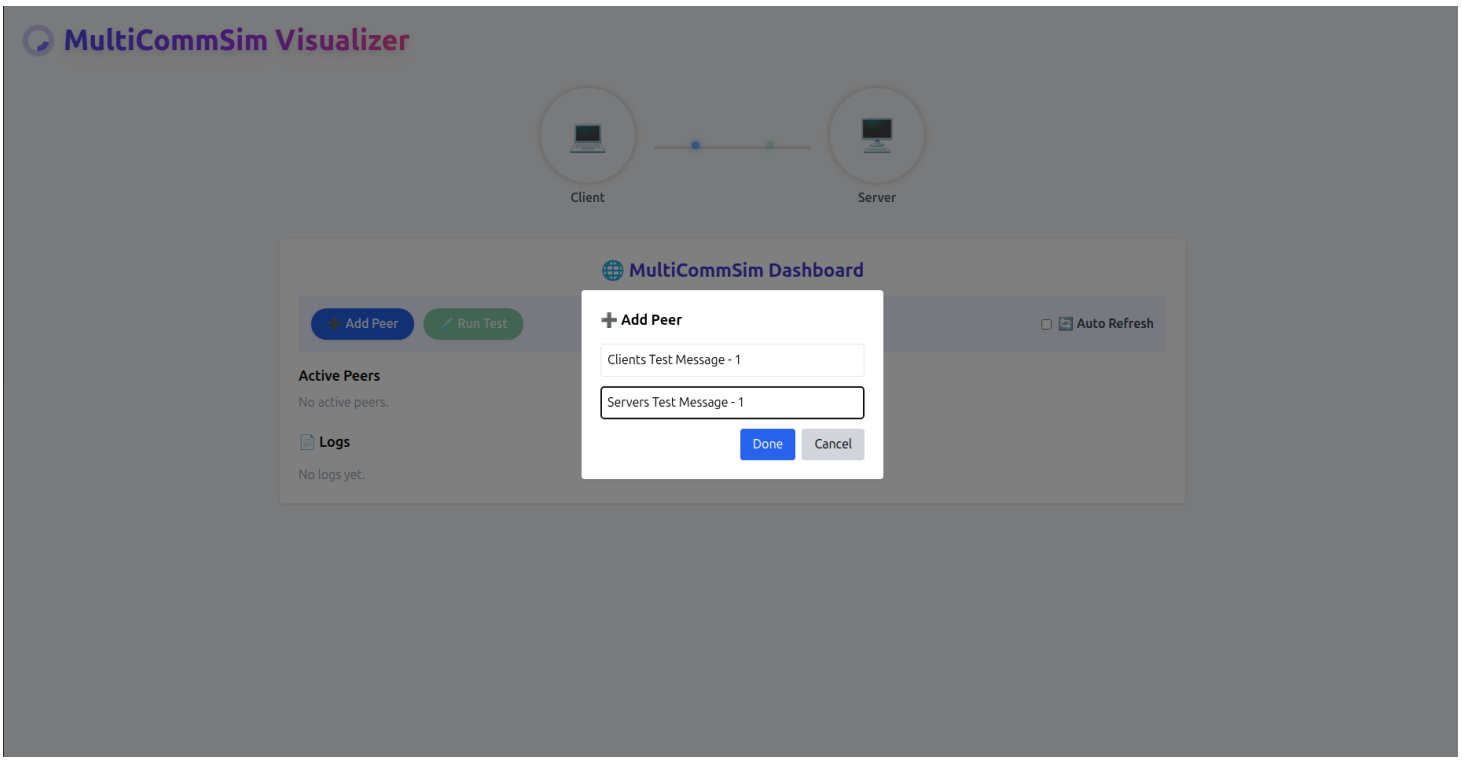


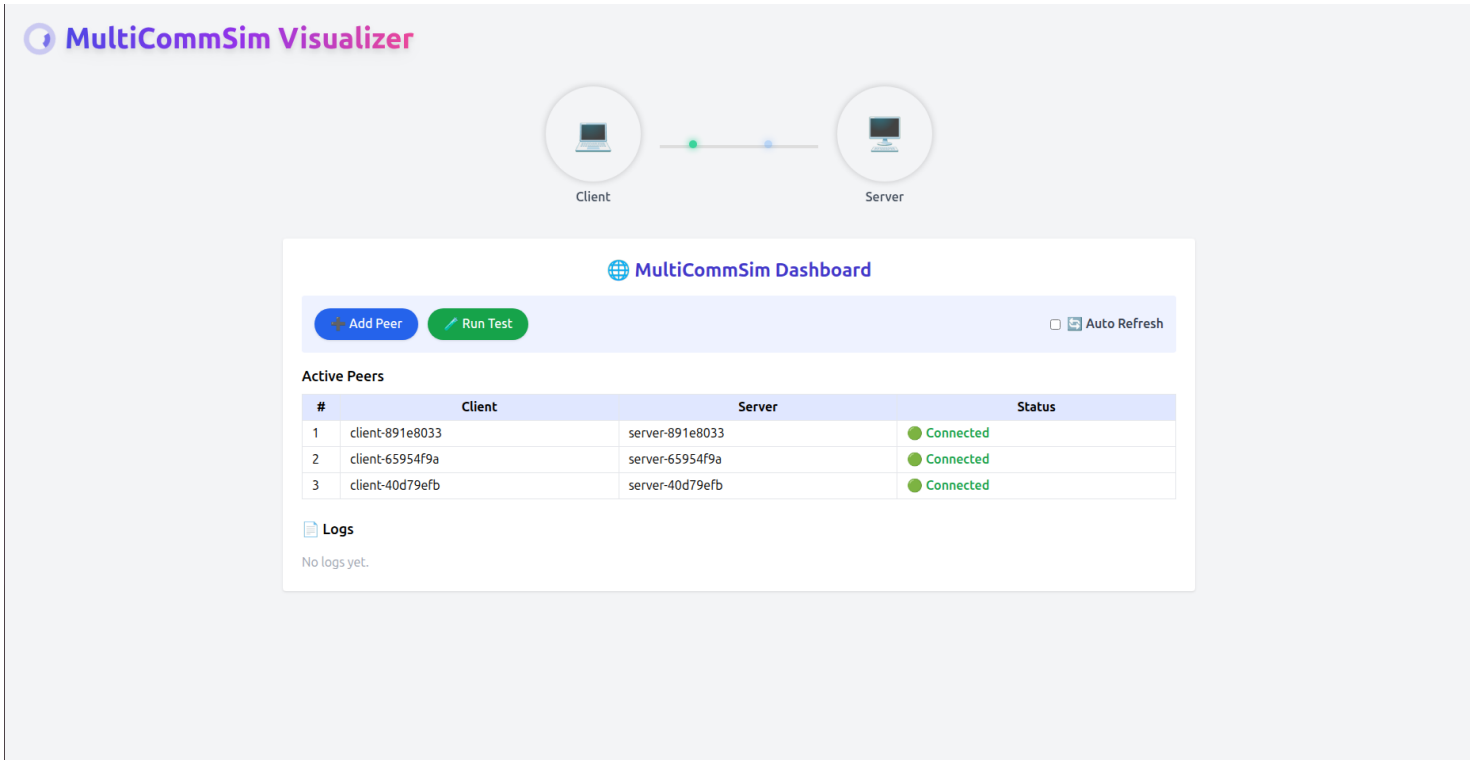
6.4 Container Deployment Diagram

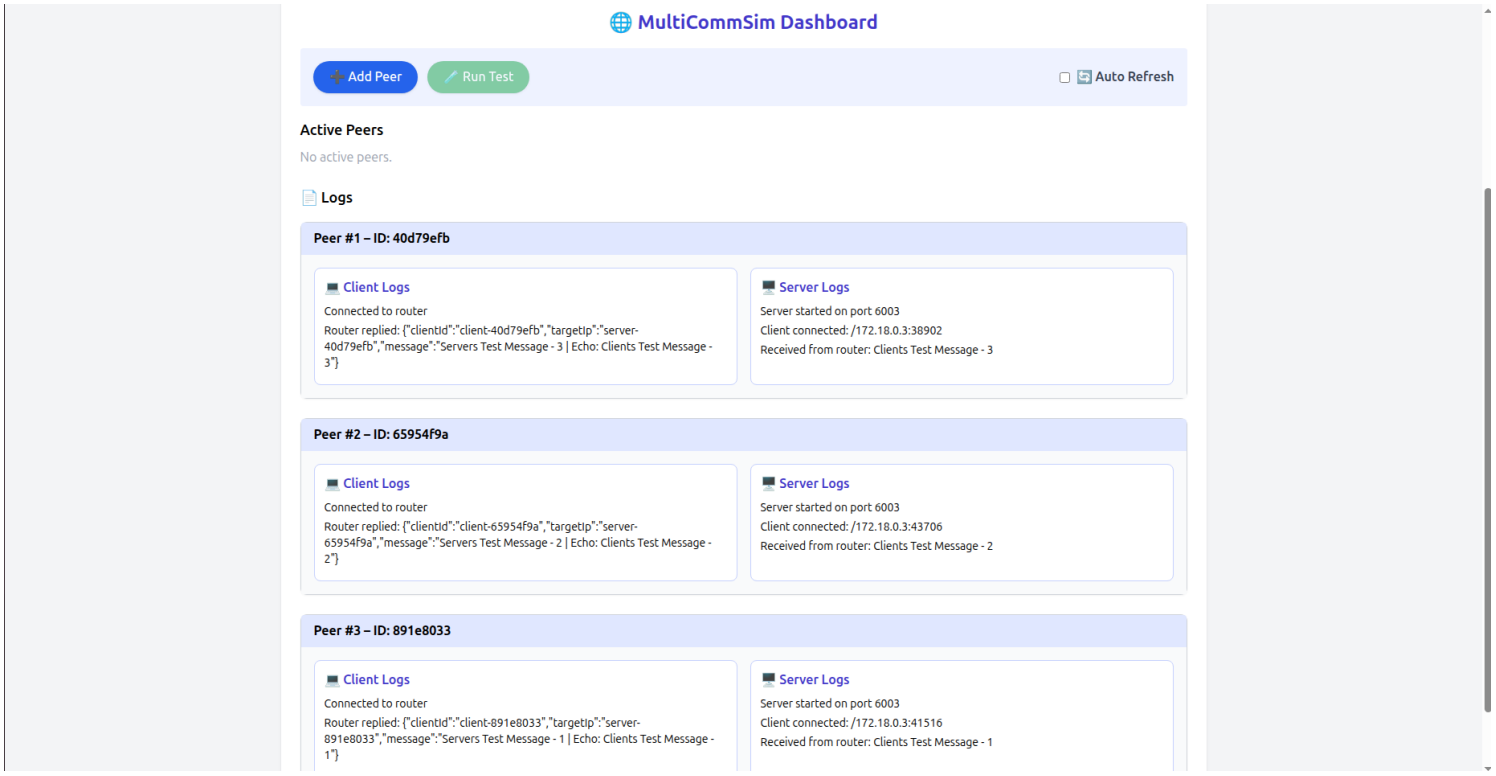


6.5 UI Images


MultiCommSim Visualizer
Client
Server
MultiCommSim Dashboard
Add Peer
Run Test
Auto Refresh
Active Peers
No active peers.
Logs
No logs yet.


MultiCommSim Visualizer
Client
Server
MultiCommSim Dashboard
Add Peer
Run Test
Auto Refresh
Active Peers
No active peers.
Logs
No logs yet.
Add Peer
Clients Test Message - 1
Servers Test Message - 1
Done
Cancel


MultiCommSim Visualizer
Client
Server
MultiCommSim Dashboard
Add Peer
Run Test
Auto Refresh
Active Peers
Client Server Status
1 client-891e8033 server-891e8033 Connected
2 client-65954f9a server-65954f9a Connected
3 client-40d79efb server-40d79efb Connected
Logs
No logs yet.


MultiCommSim Dashboard
Add Peer
Run Test
Auto Refresh
Active Peers
No active peers.
Logs
Peer #1 – ID: 40d79efb
Client Logs
Connected to router
Router replied: {"clientId":"client-40d79efb","targetIp":"server-40d79efb","message":"Servers Test Message - 3 | Echo: Clients Test Message - 3"}
Server Logs
Server started on port 6003
Client connected: /172.18.0.3:38902
Received from router: Clients Test Message - 3
Peer #2 – ID: 65954f9a
Client Logs
Connected to router
Router replied: {"clientId":"client-65954f9a","targetIp":"server-65954f9a","message":"Servers Test Message - 2 | Echo: Clients Test Message - 2"}
Server Logs
Server started on port 6003
Client connected: /172.18.0.3:43706
Received from router: Clients Test Message - 2
Peer #3 – ID: 891e8033
Client Logs
Connected to router
Router replied: {"clientId":"client-891e8033","targetIp":"server-891e8033","message":"Servers Test Message - 1 | Echo: Clients Test Message - 1"}
Server Logs
Server started on port 6003
Client connected: /172.18.0.3:41516
Received from router: Clients Test Message - 1

7. Conclusion and Evaluation

TÜBİTAK UZAY MultiCommSim projesi, çoklu peer haberleşmesini destekleyen, dinamik container yönetimi ile ölçeklenebilen, gerçek zamanlı loglama ve görsel takip imkânı sunan güçlü bir iletişim simülasyon ortamı sunmuştur.

Projenin Başarıyla Karşıladığı Özellikler:

- Peer oluşturma süreci tamamen otomatiktir (Flask API + Docker SDK)
- Peer'ler arasında TCP haberleşme Router Servisi üzerinden sağlanır
- React UI ile anlık log ve durum izleme arayüzü
- NGINX Reverse Proxy ve WebSocket opsiyonlarının yerine daha sade bir çözüm
- Docker izolasyonu sayesinde modüler sistem

Mimari Avantajlar:

- Tek IP ve tek port (**6003**) ile tüm peer'lerin yönetimi
- Router üzerinde session-based TCP yönlendirme
- React + Flask + Java + Docker dörtgeninde esnek yapı
- Peer başına özel loglama ve otomatik silme

Prepared by: Ediz Arkın Kobak

Date: 10.07.2025